FlacIO: Flat and Collective I/O for Container Image Service

Yubo Liu, Hongbo Li, Mingrui Liu, Rui Jing, Jian Guo, Bo Zhang, Hanjun Guo, Yuxin Ren, and Ning Jia, *Huawei Technologies Co., Ltd.*

https://www.usenix.org/conference/fast25/presentation/liu-yubo

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings of the 23rd USENIX Conference on File and Storage Technologies is sponsored by

FlacIO: Flat and Collective I/O for Container Image Service

Yubo Liu, Hongbo Li, Mingrui Liu, Rui Jing, Jian Guo, Bo Zhang,
Hanjun Guo, Yuxin Ren, and Ning Jia
Huawei Technologies Co., Ltd.

Abstract

This paper examines the I/O bottlenecks in the container image service. With a comprehensive analysis of existing solutions, we reveal that they suffer from high I/O amplification and excessive network traffic. Furthermore, we identify that the root cause of these problems lies in the storage-oriented and global-oriented container image abstraction. This work proposes a memory-oriented and service-oriented image abstraction, called runtime image, which represents the memory state of the root file system of the container service. The runtime image enables efficient network transfer and fast root file system construction. We design and implement FlacIO, an I/O accelerator based on the runtime image for container image service. FlacIO introduces an efficient runtime image structure that works in conjunction with a runtime page cache on a host node to achieve efficient image service. Our evaluation shows that FlacIO reduces the container cold startup latency by up to 23 and 4.6 times compared to existing full image and lazy loading solutions, respectively. In real-world applications, FlacIO achieves up to 2.25 and 1.7 times performance speedup over other systems in the object storage and machine learning training scenarios, respectively.

1 Introduction

Container technology has been widely deployed in cloud scenarios, such as elastic computing, dynamic service expansion, and disaster recovery, due to its lightweight and easy deployment advantages. The quality of the container image service is an important indicator of cloud products. Container cold startup refers to the process of loading the image from the registry node to the host node and starting the service. However, the inherent cold startup solution is to load the full image to the host node, which can cause unacceptable latency, especially for large container images [15, 18].

The key bottleneck of full image loading is the severe I/O amplification, which can reach up to dozens of times the data required for container startup according to our experiments and other related works [3, 31, 40]. There are two complementary technical routes to alleviate this bottleneck. The first route is to accelerate the image loading process. Lazy loading is the mainstream solution in this route, which allows containers to be started immediately after obtaining image metadata but load data on demand while running. Lazy loading is widely used in the production environment of cloud vendors. Typical systems include CRFS [14], Nydus [25], and DADI [18]. The second route is to migrate the cold startup out of the critical path of service launching. Typical solutions use caching/sharing [2, 11, 20, 21, 26], forking [8, 29, 36], and P2P loading [7] to launch the target container from the local/other host node.

The optimizations mentioned above do not completely solve the cold startup problem. The mitigation-based optimizations need to consume the hardware resources (*e.g.*, DRAM, network) on the host nodes, and the limited resources make a certain proportion of containers still need to be cold started under real-world workloads [31].

However, the acceleration-based optimizations, *i.e.*, lazy loading, are suboptimal. According to our quantitative analysis, although existing lazy loading solutions avoid the high latency of pulling full images, they incur high network overhead during the on-demand loading phase (up to 90% of the total overhead). Furthermore, our analysis concludes that the network overhead is mainly due to two reasons: 1) High I/O amplification (up to 3.1 times) caused by the mismatch between the access and on-demand loading granularity. 2) Massive network traffic (hundreds of thousands of packets) caused by random access behavior on image data.

Motivation. This work aims to propose an optimization that stacks on the lazy loading systems to tackle their bottlenecks. We find that traditional image abstraction based on *Storage-Oriented* (*i.e.*, recording the disk state) and *Global-Oriented* (*i.e.*, one image for multiple services) is the root cause of inefficient lazy loading. This makes the I/Os of image data loading difficult to be optimized. The main work of lazy loading can be considered as creating the root file system for the container startup. Because the data required for the same container service to start is deterministic, our motivation is to use a new image abstraction based on *Memory-Oriented* (*i.e.*, recording the memory state) and *Service-Oriented* (*i.e.*, one image for one service) for container's root file system, called *Runtime Image*, to significantly reduce the amount of network I/Os and data required for root file system construction.

Challenges. Optimizing lazy loading based on runtime image is non-trivial, and it encounters two challenges. First, the organization of runtime image needs to be carefully designed, taking into account the loading efficiency, space overhead, and container ecosystem compatibility. Second, an I/O stack for runtime image needs to be designed to support the root file system built from the runtime image while remaining lightweight. This paper designs FlacIO (**FLA**t and **C**ollective **I/O**), an I/O accelerator to enable runtime image in mainstream lazy loading systems. It includes two key designs:

1) Runtime Image. FlacIO allows container users to create and manage runtime images for specified container services from the base (original) image through a simple set of APIs. For runtime image creation, FlacIO starts the container and uses probe-based I/O tracing to accurately collect I/O on the root file system, which is then sent to the registry node for asynchronous runtime image generation. In the runtime image, related data in the I/O trace is extracted from the base image, deduplicated, and stored in continuous space, where the data is indexed by the file index and page tables built into the runtime image. During cold startup, FlacIO detects whether to use runtime image (if any) or roll back to lazy loading. By this design, runtime image loading is only charged a few small I/Os and a large I/O to obtain the image description and image data, respectively, which significantly improves network efficiency during cold startup.

2) Runtime Page Cache (RTPC). FlacIO proposes RTPC, a specific page cache in the kernel of the host node to allow building the root file system based on the runtime image. The RTPC is embedded into the OverlayFS [28] and stacked on the traditional VFS page cache. It provides a set of new OS primitives for the container platform to build the root file system, which only needs to inject (copy) the runtime image into the kernel and mount the root file system on it. Because runtime images are based on services, RTPC supports incremental loading and injection of runtime images created from the same base image. When a root file system is mounted to the RTPC, the file access is directly processed by the RTPC if the target data is hit in it. Otherwise, the access is redirected to the native VFS. As a result, RTPC ensures a lightweight root file system creation and efficient memory usage.

We implement FlacIO and adapt it to two popular lazy loading systems, CRFS [14] and Nydus [25]. Our evaluation shows that FlacIO reduces the cold startup latency by up to 4.7 times in a variety of typical container services, compared to the state-of-the-art lazy loading systems. With real-world applications, FlacIO achieves up to 2.25 and 1.7 times performance improvement over other tested systems in the object storage and ML training scenarios, respectively. In a cluster-wide container auto-scaling scenario, FlacIO delivers up to 55% faster scaling than existing solutions.

The contributions of this paper include:

- It quantitatively analyses the I/O bottleneck in container image services and observes that the bottleneck is caused by frequent small network I/Os and complicated I/O stack.
- It proposes a new image abstraction accompanied by an end-to-end solution, including the techniques of runtime image organization/management and runtime page cache.
- It adapts FlacIO to the mainstream container image service solutions (CRFS and Nydus) and demonstrates the benefits via micro benchmarks and real-world applications.

The rest of this paper is organized as follows: Section 2 and Section 3 introduces the background and motivation; Section 4 presents the key designs and implementation of FlacIO; Section 5 shows the detailed evaluation of FlacIO; Section 6 concludes the paper.

2 Background

2.1 Container Image Service

Container technology has become the infrastructure for large-scale distributed scenarios. As an important part, the image service has a significant impact on the overall performance. In the production environment, the storage stack of the container image service consists of two components: a remote registry for storing and managing images, and a storage driver on the host node for pulling the image and mounting the root file system of the container. A container image consists of multiple read-only layers at the bottom and a writable layer at the top. The storage driver uses the OverlayFS [28] to stack the files in the image and present the namespace of the root file system for the container.

Cold startup is the process of pulling the image from the remote registry over the network and launching the container service on the host node. Loading the full image to the host node is a native solution for cold startup. However, the full image loading subjects to long latency and low network bandwidth utilization because only a small amount of data in the image needs to be used during container service startup [15]. Currently, there are two complementary optimizations for inefficient full image loading:

1) Cold Startup Acceleration. Lazy loading is the mainstream mechanism to accelerate container cold startup, which has been widely used in production environments. Inspired by the fact that only a small part of the image data is required for container startup, lazy loading allows containers to be launched only after image metadata is loaded, and image data is loaded on demand during container running. CRFS [14] is a FUSE [32] file system that introduces an image format called stargz [10] to support lazy loading at the file granularity; Nydus [25] treats all layers of the image as an independent EROFS [12] and leverages FS-Cache [1] to support lazy loading at the chunk (*e.g.*, 64KB) granularity; DADI [18] proposes a block-level lazy loading solution by using the overlayered iSCSI block device; Slacker [15] maps image layers to the snapshots of remote storage (*e.g.*, Ceph [37], VMstore [13]) for lazy loading.

However, image data miss during container running will trigger I/O and block container services. Some lazy loading systems use prefetching to reduce the impact of this problem to some extent, *i.e.*, loading high-priority data before the container is launched. For example, CRFS [14] allows users to manually prioritize files during image creation for prefetching; DADI [18] uses `blktrace` to record the block I/Os during container startup on its customized virtual block device and uses `FIO` to prefetch data.

2) Cold Startup Mitigation. Orthogonally to cold startup acceleration, some works resort to the migration-based ap-

proach, which aims at reducing the frequency of cold startups. These works can be divided into several types: Caching/Sharing – FaaSCache [11], SEUSS [2], and FlashCube [21] keep hot containers alive to exploit locality; SOCK [26] and Pagurus [20] allow multiple services to share idle containers to simplify container startup; RainbowCake [38] combines the caching and sharing solutions to use their respective advantages; FaaSNet [34] provisions serverless function containers in a decentralised and scalable manner. Fork – Catalyzer [8] and Mitosis [36] resort to the customized fork mechanisms to start containers from other related processes. P2P Loading – Dragonfly [7] allows loading images from adjacent host nodes to avoid accessing the slow registry.

Cold Startup Bottleneck. The efficiency of cold startup can significantly affect the overall performance of container-based productions. For example, the long cold start latency (minute-level) of AI containers affects the QoS guarantee in our cloud product; The slow cold startup causes the long tail latency challenge in the serverless platforms of public clouds [31, 35]. However, neither the acceleration-based nor migration-based solutions effectively solve the cold startup bottleneck.

On the one hand, existing acceleration-based solutions are suboptimal. The native lazy loading suffers from high network overhead. Even if the prefetching optimizations are used, they simply replay the I/Os, which only changes the I/O priority without significant relieving network traffic. On the other hand, existing mitigation-based solutions are not applicable to all scenarios. First, they occupy resources of the host cluster (*e.g.*, memory and network), which may cause unpredictable performance jitter of online services in the resource-starved scenario. Second, they struggle with security in multi-tenant environments. To this end, this work focuses on the acceleration-based solution, but it is complementary to existing migration-based optimizations.

2.2 Lazy Loading

Flow (a) and (b) in Figure 1 show the I/O paths for file-level and block-level lazy loading, respectively. Consider the file-level solution, which contains two core components:

OverlayFS. The root file system is built upon the OverlayFS [28]. It is stacked on the local file system and contains four layers: the lower layer stores read-only files and directories of the container image; the upper layer stores modifications to the lower layer; the work layer temporarily stores intermediate states of file system operations; the merge layer integrates the upper and lower layers to provide the container with a unified view of the file system. All file accesses on the root file system are redirected to the underlying local file system by the OverlayFS.

Loading Module. When the page to be accessed is missed, the local file system notifies (by FUSE or FS-Cache) the lazy loading module to load data from the registry node. To reduce network overhead, existing solutions prefetch data to the lazy

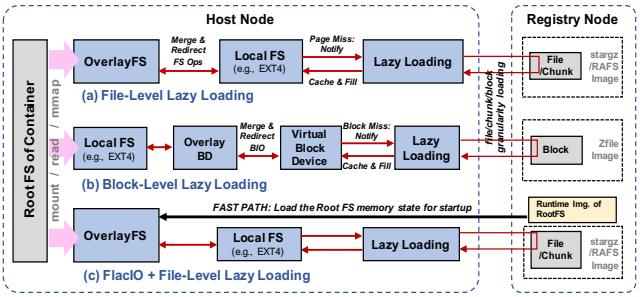

Figure 1: Architecture Comparison. File-level and block-level lazy loading solutions fetch image data on demand, which suffers from high network overhead and complex I/O path. FlacIO provides a fast path stacked on lazy loading mechanism.

loading (perfetch) cache and then populate the VFS page cache of the local file system with missing pages.

The basic mechanism of the block-level solution (DADI) is the same as that of the file-level solution. The main difference is that the overlay feature and the prefetch cache is built at the block layer. This design allows the root file system to be built on top of any local file system.

The container cold startup can be divided into three stages: *1) Deploy* stage obtains the metadata and data required for building the container runtime from the registry; *2) Running* stage creates the container runtime (*e.g.*, cgroup) in the host node; *3) Ready* stage launches the service and executes the entrypoint in the container. The container can provide services normally after these three stages are performed. Five systems covering three different technical routes are used for our analysis, including full image loading, lazy loading (CRFS [14], Nydus [25], DADI [19]), and lazy loading with prefetching (DADI-Trace [6]). We start the Pytorch container in cold and use `import torch` as the entrypoint in the experiment. The host and the registry of the testbed run on two nodes, which are interconnected through a TCP/IP network. There are two important observations that are found from the experiment:

Observation 1: *Existing lazy loading solutions reduce the latency of the "Deploy" stage, but they surfer from high overhead in the "Ready" stage.*

This observation is reflected in the "Latency Breakdown" column in Table 1. Using the full image loading as the baseline, the lazy loading solutions (CRFS, Nydus, DADI(-Trace)) shorten the latency of the "*Deploy*" stage by more than 98%, but increase the latency of the "*Ready*" stage by more than 9 times. This is because lazy loading offloads heavy I/Os when the container executes its entrypoint (*i.e.*, `import torch` in this case). The core benefit of lazy loading comes from reducing the amount of data loaded, which results in about 80% reduction in the total cold startup latency compared to the full image loading. This observation reveals the focus of lazy loading optimization, *i.e.*, inefficient I/Os in the "*Ready*" stage. Compared with the existing lazy loading solution, FlacIO reduces the latency of the "*Ready*" stage by more than 70% while ensuring the low overhead of the other stages.

Table 1: Performance Breakdown of Container Cold Startup.

Solution	Latency Breakdown				I/O Behavior	
	Deploy	Running	Ready	Total	I/O Amp.	Net. Pkg.
Full Image	124.6s	1.6s	1.7s	127.9s	47.5X	573K
CRFS	1.8s	1.2s	24.1s	27.1s	1.8X	99K
Nydus	0.8s	2.9s	21.4s	25.1s	1.6X	90K
DADI	0.6s	2.6s	17.0s	20.2s	3.1X	171K
DADI-Trace	0.7s	2.2s	17.1s	20.0s	3.0X	166K

Observation 2: *Severe I/O amplification and inefficient network accesses are the main performance bottlenecks of container cold startup.*

This observation is reflected in the "I/O Behavior" column in Table 1. On the one hand, although existing lazy loading solutions significantly reduce the I/O amplification compared to the full image loading solution due to the on-demand loading, they still suffer from 1.6 to 3.1 times I/O amplification at the page level. The main reason is that they use a relatively large loading granularity (*e.g.*, file, chunk), but the locality of the image data at startup is not strong. On the other hand, existing lazy loading solutions trigger a large amount of random access to the remote image data in the "*Ready*" stage, resulting in low utilization of network resources (hundreds of thousands of packets are required for each cold startup). In addition, I/O amplification wastes network bandwidth.

The combination of low I/O amplification and efficient network accesses is a dilemma for existing lazy loading solutions – smaller on-demand loading granularity reduces I/O amplification but increases network load, and vice versa. Prefetching high-priority data is a way to alleviate this contradiction, but it is still suboptimal. In the tested systems in Table 1, CRFS prefetches the full image in the background, while DADI-Trace prefetches the data in the block I/O trace of historical the cold startup. However, experimental results show that these methods do not solve the problem well because they do not aggregate I/O efficiently and do not prefetch accurately. For example, DADI-Trace cannot precisely set the trace window and simply uses I/O replay to prefetch. In contrast, FlacIO only takes 1.1 times amplification and 22K network packets in this experiment.

3 Motivation: A New Image Abstraction

The main goal of lazy loading is to build the root file system in the host node for the container. We argue that the traditional image abstraction is a key factor in the inadequacy of lazy loading described above, it is: ***Global Oriented*** – It holds a complete set of files/namespaces required for running a certain type of services. However, each service uses only a small portion of the full file data, and the requirements vary from service to service. ***Storage Oriented*** – It records the disk layout of the root file system, thus, lazy loading needs to map the I/O to the disk layout, obtain data from the registry, and rebuild the local memory state (filling namespaces/indexes/data) before being used by the container.

This abstraction brings many problems in the container cold startup. 1) I/O is difficult to aggregate because the data required during container startup is discretely distributed in different locations across different image files. 2) I/O amplification is hard to eliminate because data is compressed and stored in images, which makes the data difficult to index at page granularity [39]. 3) I/O locality is difficult to optimize because of the differences between services. 4) Lazy loading requires complex I/O forwarding to load data and build the memory state of the root file system.

The motivation of this work is to make the image abstraction to be ***Memory Oriented*** – recording the memory state of the container, and ***Service Oriented*** – one image for one service. To this end, we pre-build the memory state of the root file system of the container service, called ***Runtime Image***. It delivers two advantages: First, it is efficient for network transfer because it contains only the smallest set of data needed to start the container and is beneficial to I/O aggregation. Second, it can support fast root file system construction because it contains a complete memory state.

As shown in Flow (c) in Figure 1, the runtime image is stacked on the lazy loading mechanism and brings a fast path to prepare the root file system through a flat and collective approach. The general-purpose full-memory state checkpoint (*e.g.*, CRIU [5, 33]) cannot be used for the runtime image because it is not co-designed with the container image service and is applicable to limited scenarios (*e.g.*, difficult to handle different architecture/tenant/security scenarios). In contrast, the runtime image is deeply integrated with the container ecosystem, and it records only the memory state of the container's root file system, which is small and stateless. However, runtime image design is a non-trivial work:

Challenge 1: How to organize the runtime image in an efficient manner based on the new image abstraction? First, it is required to accurately record the minimum set of metadata and data required for a cold startup. Second, it must be compact so as not to put a heavy burden on backend image storage. Third, it has to be smoothly embedded into mainstream container runtimes (*e.g.*, Containerd [4]) and transparent to the upper-layer systems and users. (§4.1)

Challenge 2: How to build a lightweight I/O stack on the host node for the runtime image? On the one hand, the current kernel does not support directly loading a definite memory state for the container's root file system, which requires new OS primitives and cache mechanism. On the other hand, the runtime image is stacked on a lazy loading mechanism, so the new I/O stack needs to be compatible with the legacy on-demand loading I/O stack. (§4.2)

4 FlacIO Design

This work proposes FlacIO (**FLA**t and **C**ollective **I/O**), a novel I/O solution for container image service. Figure 2 shows the architecture of FlacIO. It includes four main components:

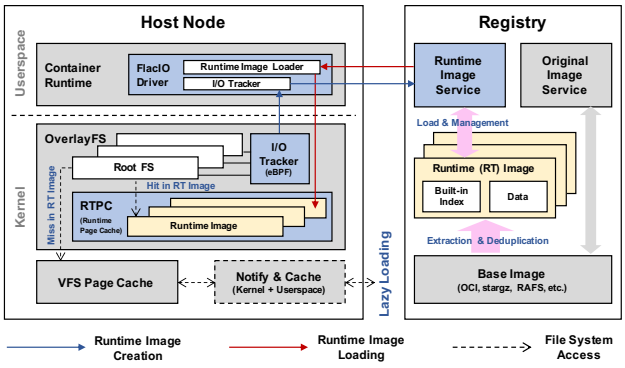

Figure 2: Architecture of FlacIO. In runtime image creation, FlacIO driver invokes the I/O tracker to collect container's I/Os and sends them to the runtime image service for asynchronous generation. In cold startup, the driver pulls and injects the runtime image into the RTPC. The root file system can be mounted on the runtime image.

1) The FlacIO driver is running in the container runtime (*e.g.*, Containerd) to play the role of the control plane. 2) The I/O tracker works in the VFS and collects I/O behavior during container running. 3) The runtime page cache (RTPC) is implemented in the OverlayFS, which provides new OS primitives to allow containers to efficiently build root file systems using runtime images. 4) The runtime image service is deployed in the image registry to serve runtime image management (*e.g.*, loading, offline generation).

APIs. A set of APIs are provided by FlacIO for the container runtime and the container end-user (Table 2) to enable the runtime image optimization in existing container ecosystems. In terms of mechanism, FlacIO can adapt to any lazy loading solution through minor engineering efforts.

Container End-User: The `rt_create` and `rt_delete` allow end-users to enable or disable the runtime image optimization for a service as required. If a runtime image exists for the container service, FlacIO will automatically start with our fast way, otherwise, the startup process will fall back to the traditional lazy loading mechanism.

Container Runtime: The `mount -rt` is a customized mounting flag of the OverlayFS, and the container runtime (*e.g.*, Containerd) can bind the root file system to the runtime image by setting the `-rt` flag with the container service ID. The `rt_diff` and `rt_inject` are used to implement incremental loading and efficient injection of runtime images.

Key Workflow. When the container end-user creates a runtime image for the container service, the FlacIO driver starts the container by the traditional lazy loading method and logs its I/O behavior until the readiness probe condition is met. Then, the I/O log is sent to the registry to generate the corresponding runtime image offline. During the cold startup of a container, the FlacIO driver checks whether the target runtime image exists in the registry. If it does not exist, the container is started using the traditional lazy loading method. If the target runtime image exists, the FlacIO driver pulls it to the host node by incremental loading and injects it into the RTPC. The container is then started in the traditional manner.

Table 2: Main APIs of FlacIO

User	API	Parameter	Description
Container End-User	rt_create	*image_name entrypoint probe*	Create a runtime image for the provided image and entrypoint, and return the *service_id*
	rt_delete	*service_id*	Delete the runtime image related to the *service_id*
Container Runtime	mount -rt	*service_id*	Build the root file system based on the runtime image
	rt_diff	*service_id dzone_bitmap*	Compare the bitmaps and return the *diff_bitmap* of the missing page for incremental loading
	rt_inject	*service_id diff_bitmap rt_meta/data*	Inject the metadata and data of the runtime image into the target runtime page cache

4.1 Runtime Image

The runtime image records the memory state of the root file system when the container service is started. It contains the minimum set of data needed to start the container and the index of that data. Runtime image relies on two key designs: 1) A probe-based tracing mechanism is designed to accurately collect the container's I/O behaviors. 2) A sophisticated runtime image structure is proposed to meet high network transmission efficiency, low storage overhead, and high container ecosystem compatibility.

4.1.1 Probe-based Container I/O Tracing

Runtime image is asked to record only the data required for container service startup, but accurate tracing of the container's I/O behavior is not easy. To make an efficient runtime image, the I/O tracker needs to address two challenges: First, how to collect the minimum set of I/Os required for container service startup? Second, how to accurately collect the I/O requests of the target container? The container probe and the file-level I/O tracker are designed to address these two challenges, respectively.

Container Probe. The I/O behavior during container running is complex, and only part of the I/Os are used to start the container service (*e.g.*, loading necessary libraries). FlacIO uses the probe-based method to enable the underlying I/O tracker to detect the I/O behavior in the container. The core principle is to stop I/O tracing when the probe captures the corresponding event. Probes are classified into external probes and internal probes. External probes detect container status outside containers. FlacIO provides some default external probes, such as network port status detection. Correspondingly, the internal probe runs in the container and is an entrypoint, and FlacIO uses the internal probe to run the container during I/O tracing. The user is required to specify

the type and content of the probe (default probe/executable external program/entrypoint) when calling `rt_create`.

For example, network services (*e.g.*, Ngnix, Httpd) are suitable for default HTTP status probes, while framework services (*e.g.*, Pytorch) are suitable for internal probes (*e.g.*, having the probe load the necessary libraries). Compared with the static tracing mechanism (manually set the duration) in DADI [18], the probe-based solution can collect I/Os required for container startup more accurately and flexibly.

File I/O Tracker. FlacIO traces I/Os at file-level rather than block-level for two reasons. First, it is compatible with different container platforms. Tracking I/Os in the root file system is suitable for both file-level and block-level lazy loading systems, while tracking at the block layer depends on the specific overlay block device. Second, file-level I/O tracing is more accurate. The block layer and container I/O requests exhibit a certain degree of discrepancy due to the I/O re-orchestration of the file system.

The container uses image data through a file system call (`read`) and memory mapping (`mmap`). The I/O tracker is designed based on eBPF [9]. We add the eBPF points in the entries of file read and page fault to ensure that I/O requests are collected accurately and completely. The I/O tracker generates an I/O trace after tracing, which includes multiple triples and each of them records the file path, offset, and size of the I/O. The I/O trace is sent to the registry for asynchronous generation of the runtime image.

4.1.2 Organization and Management

Runtime images are service-specific, which requires that FlacIO uniquely represent the container service. The base (original) image and the entrypoint can reflect the container environment and the behavior of the service, respectively. FlacIO generates a service ID based on the hash value of the base image name and the entrypoint. However, a base image may correspond to multiple services, and the I/Os required for these services to start may be highly duplicated. To reduce the footprint, we group the runtime images created on the same base image. Figure 3 shows the organization of the runtime image, and it includes three parts:

Group Metadata. It contains metadata that is global to all runtime images (services) in the group. The runtime list is used to index the runtime images through the service IDs. To support data deduplication inside the group, a fingerprint index is used to store the SHA256 value for each page in the group and their location in the group data zone. It also contains an allocator for managing the group data zone space.

Service Metadata. Each runtime image in the group has its metadata. The file index describes the namespace of the root file system, and each file handle points to a page table that is used to index pages on the group data zone. In addition, service metadata contains a bitmap that records the distribution of the runtime image's pages in the group data zone, which will be used during incremental loading.

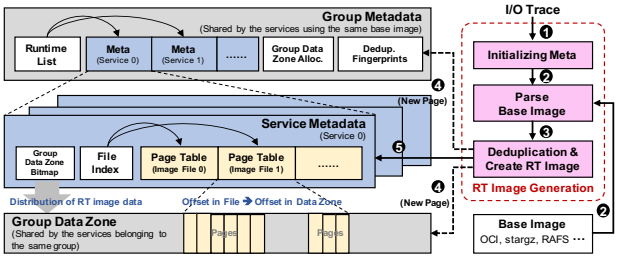

Figure 3: Runtime Image Layout. Services based on the same base (original) image form a group, each with global metadata and a shared group data zone. Each service has independent metadata for indexing data in the group data zone.

Group Data Zone. It is a continuous storage space for storing runtime image data of the group. The pages belonging to the same group are deduplicated. Deduplication within the group is reasonable because based on our analysis of real-world workloads, the duplication of runtime data across different base images is less than 1%. In addition, the size of the group data zone is dynamically adjusted with the addition or deletion of the runtime image.

Generation and Deletion (offline). Figure 3 shows the process of runtime image generation in the registry: ❶ When the generator receives an I/O trace, it initializes the relative group and service metadata by using the service ID. ❷ Next, the generator parses out the pages contained in the record from the base (original) image. The image parsing module can support different types of images (*e.g.*, OCI, stargz, RAFS) by adding image format parsers. ❸ The deduplication module calculates the fingerprint (by SHA256) of the page obtained in the previous step, and checks whether the same fingerprint exists in the fingerprint index. ❹ If the fingerprint does not exist, the generator allocates a new page in the group data zone, copies the page to it, and updates the fingerprint index. ❺ The page location of the group data zone is recorded in the related page table, and the related bit in the bitmap of the group data zone is set to 1 (if it is a new page).

The runtime image is deleted in three cases: 1) The entrypoint of the service is changed; 2) The base image is deleted; 3) Users explicitly invoke the runtime image deletion interface provided by FlacIO. The deletion process examines the fingerprint of each page in the target runtime image and decrements its reference counting by one. When a page reference is 0, the fingerprint is deleted and the allocator reclaims its space. Then, the service metadata and the target runtime image information in the group metadata will be deleted.

4.2 Runtime Page Cache

FlacIO proposes RTPC (runtime page cache), a special page cache subsystem to provide a lightweight I/O stack in coordination with the runtime image. It includes three key designs: 1) A new cache framework built in the OverlayFS to allow the root file system of the container to be built from the run-

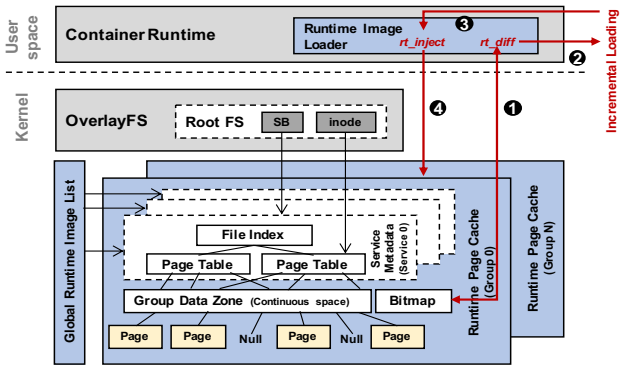

Figure 4: Runtime Page Cache (RTPC). It is built in the OverlayFS and stacked on the VFS page cache. The runtime images that belong to the same group share a RTPC.

time image. 2) An incremental loading mechanism is used to further improve the efficiency of runtime image loading. 3) A set of file access logic based on RTPC.

4.2.1 Framework and Operations

Figure 4 shows the framework of RTPC. The runtime images that belong to the same group share a RTPC. Similar to the structure of the runtime image group, each RTPC has multiple service metadata and a group data zone. The structure of the service metadata includes the file index and multiple page tables, which are loaded from the service metadata of the runtime image. The group data zone contains a contiguous kernel memory address space used to store data of the runtime images. The logic size of the group data zone of the RTPC is the same as it in the runtime image. However, the physical space for the group data zone is not allocated until its associated runtime image is loaded. Each RTPC maintains a page-granularity bitmap to record which pages in the group data zone have been loaded (used for incremental loading). In addition, the global runtime image list records all loaded runtime images from which the root file system can be found and mounted.

Incremental Runtime Image Loading. This process ensures that data located in the same runtime image group is not redundantly loaded. The FlacIO driver is responsible for the incremental loading, and it relies on two new OS primitives exposed by FlacIO: `rt_diff` is used to detect which pages of the target runtime image need loaded; `rt_inject` is used to inject a given runtime image into the OverlayFS.

Figure 4 shows the main process: ❶ Obtain the service metadata of the target runtime image from the registry and call the `rt_diff` with the bitmap in the service metadata as a parameter to obtain the positions of the pages that are not loaded in the RTPC. The *diff_bitmap* will be returned from this step. ❷ Send the *diff_bitmap* to the registry to request data. ❸ Pull the missed runtime image data from the registry. The runtime image service traverses the *diff_bitmap* and aggregates the missed pages into a single network I/O. ❹ Call the `rt_inject` to load the service metadata (file index and page tables) and the missed pages to the RTPC. Two new OS primitives provided by RTPC are described below.

***Primitive 1:* `rt_diff`.** The input parameters of this function include the service ID (*service_id*) and the group data zone bitmap obtained from the service metadata of the runtime image (*dzone_bitmap*). The `rt_diff` will find the corresponding RTPC through the *service_id*, and the target RTPC will be created if it does not exist. The bitmap of the RTPC is then compared with the input *dzone_bitmap* to find the pages in the target runtime image that have not yet been loaded into the RTPC. Finally, the function returns a bitmap of the missed pages (*diff_bitmap*) for the runtime image loader (in the FlacIO driver).

***Primitive 2:* `rt_inject`.** The input parameters of this function include the service ID (*service_id*), the differential bitmap (*diff_bitmap*), the service metadata (*rt_meta*), and the missed data loaded from the registry (*rt_data*). The process of `rt_inject` includes three steps: 1) It searches the target RTPC by using the *service_id*. 2) It copies the service metadata and the missed data of the runtime image to the RTPC. In particular, when injecting data, `rt_inject` copies the data from the *rt_data* buffer to their specified location on the group data zone based on the *diff_bitmap*. Note that the copy overhead can be further optimized with existing zero copy techniques [22]. 3) It updates the bitmap of the RTPC and exposes the new runtime image to the global list for root file system mounting. The association between the root file system and the runtime image is by mounting.

4.2.2 File Operations on RTPC

RTPC is implemented by hooking the corresponding file interfaces of OverlayFS. It is integrated into the existing VFS file access logic and is transparent to file system users. RTPC is a read-only kernel cache, which handles the `open`, `read`, and `mmap` operations on the files and pages in the runtime image. Accesses to files and pages that are not part of the runtime image is handled by the original VFS process.

Mount & Unmount: When the container is started with FlacIO, the container platform uses the `-rt` flag to `mount` the root file system. With the service ID (*service_id*) passed in the mount parameter, RTPC associates the corresponding runtime image to the super block of the root file system. After that, file accesses on that root file system can be checked in the super block to determine whether to enter the processing logic of RTPC. The `unmount` is the same as the traditional VFS process, except that the corresponding RTPC (if any) is marked as idle and taken over by the cache policy.

File Open: When a file in the runtime image is opened, RTPC gets the corresponding service metadata by using the pointer recorded in the superblock of the root file system. RTPC searches the file (by the hash value of the path) in the file index in the service metadata. If the file exists, the RTPC points the *i_private* in the inode to the corresponding page

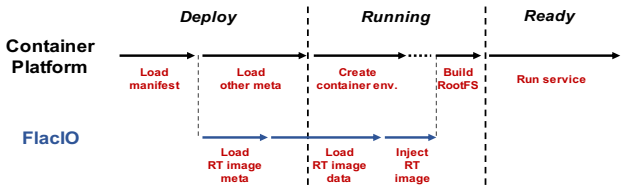

Figure 5: Workflow of Container Service Startup. FlacIO is compatible with mainstream container ecosystems. The runtime image loading process can be executed concurrently with most other container startup processes.

table in service metadata. If the file does not exist in the file index, the open operation falls back to the traditional flow.

File Read: The `read` operation determines whether the inode is associated with the RTPC by the *i_private* pointer. If not (*i.e.*, pointer is null), the read operation is rolled back to the traditional VFS read process. Otherwise, the `read` operation searches the page table in the service metadata based on the access offset. If the page exists, the data is read from the RTPC, otherwise, the `read` operation falls back to the traditional VFS process.

File Map: The `mmap` operation in RTPC is the same as the traditional process. If a memory address is accessed and it is not mapped to any physical page, a page fault is triggered. We customize the page fault mechanism to handle the case where the target page is on the RTPC. The RTPC page fault mechanism uses the *i_private* pointer in the inode to check whether the file is associated with the RTPC. If yes, the corresponding page in the group data zone is found and its physical address is mapped to the page fault location. Otherwise, the page fault rolls back to the traditional process. In addition, the RTPC page fault also supports the optimizations in the traditional page faults (*e.g.*, pre-fault).

Performance Analysis of File Access. RTPC is stacked on VFS and therefore introduces additional logic in file access operations, but they have minimal impact on the overall performance. For data that resides in the RTPC, this type of access even outperforms traditional VFS because of the small size of the runtime image that makes data indexing efficient. For data that resides in the VFS page cache but is stacked by the RTPC, file access needs to be forwarded. However, the detection logic is lightweight (lookup in a small hash table), so this overhead is almost negligible. We evaluate these effects in the experiments in §5.2.5.

4.2.3 Cache Policy

FlacIO keeps the runtime image in the memory for the entire lifetime of the container. At the same time, FlacIO allows recently used runtime images to be cached on the host node to improve startup efficiency in strong locality scenarios. The maximum memory space used for RTPC can be configured by the user. During each runtime image loading, FlacIO determines whether the RTPC size exceeds the threshold. If yes, FlacIO attempts to evict the cached runtime images in the FIFO manner (other advanced algorithms also apply [23, 30]). When a container is destroyed, the corresponding root file system (if has) is unmounted and its associated RTPC space is reclaimed. If the space is still insufficient, *e.g.*, all runtime images have running containers or the reclaimed space is smaller than the incoming runtime image, the container startup process rolls back to the lazy loading mechanism.

4.3 Putting Everything Together

4.3.1 Implementation

In the FlacIO prototype, the RTPC and the FlacIO driver are implemented in the OverlayFS and the Containerd, respectively, which are fundamental components of mainstream container platforms with lazy loading. The runtime image management is a standalone service deployed in the registry node. To this end, the container platform is implemented based on the OverlayFS and the Containerd (*e.g.*, CRFS, Nydus) can directly enable FlacIO without additional code changes. However, we believe that the design of FlacIO is universal and can be adapted to any lazy loading system (*e.g.*, block-level lazy loading). In our prototype, 188LOC and 182LOC are required for porting FlacIO to CRFS and Nydus, respectively. The core implementation is detailed below.

Host Kernel: I/O tracker is implemented by adding the eBPF probes into the I/O path of the container startup (`read`, `mmap`, and page fault) to collect and generate the I/O trace. RTPC is implemented in the OverlayFS, which exposes the `rt_diff` and `rt_inject` primitives for userspace through the *sysfs*. In addition, part of the file operations of the OverlayFS are redirected to the process logic of RTPC.

Containerd: FlacIO driver is implemented as a plug-in module in the snapshotter of the Containerd. It is used to bridge the runtime image service, the I/O tracker, and the RTPC. FlacIO driver provides the interfaces of `rt_create` and `rt_delete` for the container end-user to build the runtime image in the probe-based manner. These APIs are implemented by encapsulating the existing interface of the Containerd. At the same time, FlacIO driver embeds the runtime image loading process into the traditional container startup process by encapsulating the RTPC interfaces.

Registry: FlacIO does not modify the normal registry service but starts a separate daemon to manage the runtime image, and it interacts with the FlacIO driver by RESTful. Currently, runtime images are stored in the file system (local or distributed), but they can also be stored in the object storage or block storage through simple interface bridging.

4.3.2 End-to-End Workflow

We demonstrate the benefits of FlacIO by putting it into an end-to-end container cold startup process. The typical file-level lazy loading process is used as an example. Figure 5 shows the end-to-end workflow.

In the "***Deploy***" stage, the container platform loads the manifest and the configuration of the target base (original) image from the registry. Since the information of the runtime image is recorded in the manifest, FlacIO can load the metadata of the target runtime image once the manifest is parsed. Because of the small size of the runtime image metadata, the loading overhead can easily overlap with other operations of the container platform.

In the "***Running***" stage, FlacIO starts to pull the image data after obtaining the metadata of the runtime image (may start before this state). The image pulling process can overlap (in whole or in part) with the container environment preparation (*e.g.*, creating cgroup). After the data of the runtime image is loaded, FlacIO injects it into the RTPC. If the container starts with FlacIO, the root file system of the container can be built only after the runtime image is injected. Fortunately, this blocking interval is short in many scenarios because of the small size of the runtime image.

In the "***Ready***" stage, the service in the container starts to initialize, which triggers a large number of I/Os in traditional lazy loading systems. In contrast, FlacIO does nothing because the data required for container service startup has been included in the runtime image, therefore, the service can complete this stage with the efficiency of the full image loading solution.

Warm Startup. Container services can be started from runtime images in the memory of the host node (cached or in use by other identical services). The container platform checks whether the target runtime image exists in the host node before startup, if yes, the metadata and data loading will be skipped. The subsequent process of warm startup is the same as that in the cold startup scenario (*e.g.*, mount and file accesses).

4.3.3 Comparison with Other Optimizations

All analyzed/evaluated systems are equipped with prefetch cache, while they use different prefetching strategies. They can be divided into three categories:

1) Expanded Prefetch: It is the inherent optimization in the lazy loading solutions (*e.g.*, Nydus [25], CRFS [14]) that simply loads extra data beyond the missed range. However, this mechanism is blind, and it relies on the I/O behavior the container to have strong locality. In fact, this approach is a negative optimization in some scenarios.

2) Prioritize Files Perferch: As the improvement of the expanded prefetch, some lazy loading systems (*e.g.*, CRFS with estargz [10] image format) allow users to provide the prioritized files list of the image, and the prefetch mechanism loads the image files in descending order of priority. This optimization relies heavily on user experience. However, upper-layer users cannot perceive the priority of image files. At the same time, the mechanism performs prefetching at the file granularity, which often leads to I/O amplification.

3) Trace Replay: DADI [18] allows users to use `blktrace` to trace I/Os during container startup and use `FIO` to replay the trace during subsequent container cold startup. As the state-of-the-arts optimization, it can theoretically load I/Os more accurately than previous solutions, but it is limited by three bottlenecks. First, the size of the I/O trace window is difficult to determine, which results in inefficient I/O replay. Second, I/O replay is difficult to achieve efficient aggregation, which affects network efficiency. Third, it relies on an independent ecosystem (block-level lazy loading) and cannot be a universal optimization solution.

FlacIO delivers the following advantages compared to related works: 1) Accurate container I/O tracing. The probe-based I/O tracing mechanism enables FlacIO to flexibly and accurately collect I/Os during the startup of different container services, significantly reducing I/O amplification. 2) High network efficiency. The runtime image is network-friendly and can significantly improve network efficiency. 3) Lightweight I/O stack. RTPC simplifies the I/O stack by allowing the memory state of the root file system to be built in place.

4.4 Discussion

Although FlacIO offers promising performance, it also encounters some challenges, which we discuss below.

Trade-offs of Runtime Image. It includes two aspects. 1) Backend storage space overhead. Thanks to accurate I/O tracing and deduplication, runtime images occupy a small percentage of storage space (about 5% of the total). 2) Generality. Maintaining per-service runtime image imposes a burden on service updates to some extent. For services that are not frequently updated, one-production for long-term benefit is a good deal. For services that are frequently updated, FlacIO can degenerate to the traditional lazy loading mechanism, which has no negative impact.

Probe-based Tracing Accuracy. Probe-based tracing collects all file I/Os on the root file system only during startup, thus having 100% accuracy (at page granularity). Consider two dynamic scenarios: 1) Unchanged services with changed running load. Container runtime I/Os are not traced, and probe events ensure external services availability (*e.g.*, HTTP-OK and library loading), thus accuracy is not affected. 2) Service (image/entrypoint) changes. Users may need to redefine probe events and trace I/Os in this scenario.

User Dependence. Users only need to define the probe in `rt_create`. FlacIO provides default probes (*e.g.*, HTTP status detection) that deliver comparable performance for most services (*e.g.*, cache and network proxy containers). For framework services, users can incorporate necessary libraries as probes. Our experiments show that these "fool-like" probes are already working well. In addition, probe-like mechanisms are widely used in clouds (*e.g.*, service status detection in Kubernetes [17]), which can be used for reference.

Memory Footprint. FlacIO can reduce the DRAM footprint of lazy loading solutions. RTPC is not an extra cache but a specialized file system page cache, so image data can

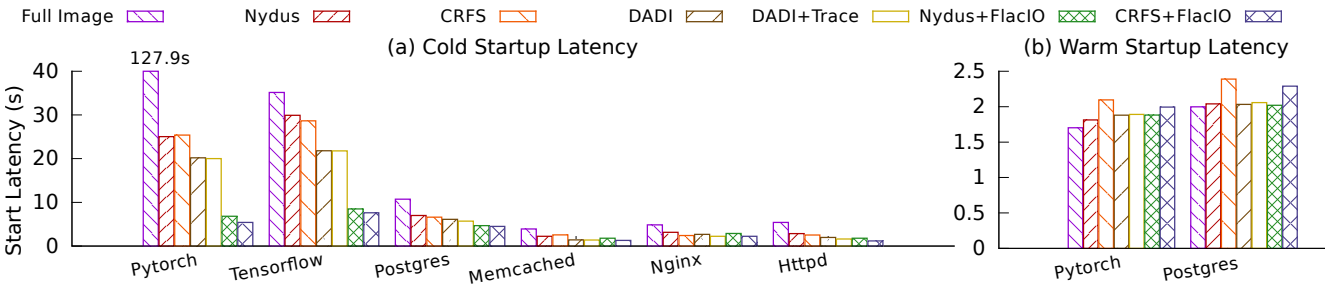

Figure 6: Cold and Warm Startup Latency.

be loaded directly from the registry into the file system page cache, saving the memory overhead of prefetch cache in existing lazy loading systems. In addition, accurate data loading also reduces the DRAM footprint. In multi-service scenarios, the runtime image deduplication and incremental loading avoid memory redundancy between services using the same base image in RTPC. For services belonging to different base images, according to our experience, the memory repetition rate does not exceed 1% (between services in Table 3).

Security. FlacIO maintains the same security level as existing containers. Upon startup, FlacIO loads data into the kernel, which provdies secure sharing via file system page cache. During runtime, FlacIO does not touch existing container security mechanisms (*e.g.*, cgroup, namespace).

5 Evaluation

All containers are run on a server with 24-core x86 CPU @ 2.30GHz, 256GB DRAM, and it is connected to the image registry through 10Gbs network. The operating system on the server is openEuler 22.03 LTS [27] with Linux 6.5 kernel. The Containerd version is v1.7.1. To demonstrate the design benefits in FlacIO, we compare FlacIO with five state-of-the-art image loading solutions: full image loading (native Containerd), file-level lazy loading (CRFS [14] and Nydus [25]), block-level lazy loading (DADI [18]), and lazy loading with trace perfetching (DADI+Trace [6]). For FlacIO, we adapted it to CRFS and Nydus and included them in all our experiments.

5.1 Container Startup Performance

We select six popular container services to evaluate the cold and warm startup latency. To get closer to real-world scenarios, we calculate the latency they can service externally. For framework services (Pytorch and Tensorflow), the completion of the startup is indicated by the successful loading of their core libraries. For daemon services (Postgres, Memcached, Nginx, and Httpd), the completion of the startup is indicated by the successful access of their HTTP ports.

We use external and internal probes for daemon and framework services, respectively. For daemon services, the tracing interval is from the time when the container is started to the time when the service is ready for processing HTTP requests. For framework services, the trace interval is from the time when the container is started to the time when the `import` (`torch` and `tensorflow`) of Python is successfully executed. We use the same probes unless otherwise specified. In DADI+Trace, we set the tracing interval to be greater than the cold startup time of the corresponding container.

5.1.1 Cold Startup Latency

Figure 6(a) shows the results. FlacIO achieves up to 4.5 times lower latency than the lazy loading solutions in cold startup, and up to 23 times latency reduction compared to the full image loading solution. The reason is that FlacIO has low network overhead, which benefits from the runtime image design to efficiently aggregate I/O and reduce I/O amplification. Furthermore, the RTPC ensures that the runtime image-based I/O stack does not refer to significant overhead. DADI+Trace is another optimization on lazy loading, which optimizes the accuracy of image data loading through replay with pre-collected I/O trace. Compared with DADI-Trace, CRFS+FlacIO improves performance by 2.7 times (Pytorch), 1.8 times (Tensorflow), 27% (Postgres), 6.5% (Memcached), 0.8% (Nginx), and 36% (Httpd). This is because DADI-Trace can improve the loading accuracy to some extent, but it still has I/O amplification and cannot efficiently aggregate network I/Os. Nydus+FlacIO is worse than DADI-Trace in some containers (Memcached/Nginx/Httpd) due to the overhead of the underlying lazy loading system (*i.e.*, Nydus *vs.* CRFS).

Through further analysis of the results, we find that FlacIO has a greater advantage in images containing a large number of files (*e.g.*, Pytorch and Tensorflow), because this makes it more difficult for lazy loading to take advantage of data locality in the load unit, increasing I/O amplification and network traffic. For instance, with Tensorflow, CRFS+FlacIO incurs 3.7, 3.9, and 2.8 times less startup latency than CRFS, Nydus, and DADI, respectively.

5.1.2 Warm Startup Latency

Warm startup means that the (runtime) image has been cached locally. We use Postgres and Pytorch to evaluate this case because other container services have similar results for warm startup. Figure 6(b) shows that the warm startup latency is similar for all systems. For the lazy loading systems, expensive on-demand loads are not triggered because the page cache

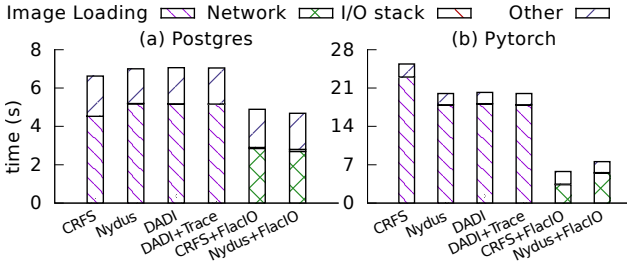

Figure 7: Performance Breakdown.

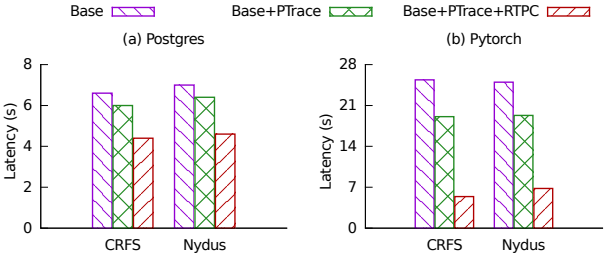

Figure 8: Factor Analysis.

of the root file system has been warmed up. For the FlacIO-based systems, FlacIO does not increase the warm startup latency. Since FlacIO is superimposed on lazy loading systems, the comparison should be the same system enabled and disabled FlacIO (*i.e.*, Nydus *vs.* Nydus+FlacIO and CRFS *vs.* CRFS+FlacIO). The main overhead of FlacIO is to mount the root file system on it and redirect the file I/Os, which is a lightweight process ensured by the RTPC. In addition, the full image loading performs the best, but the difference is only a millisecond-level, which is negligible in many scenarios.

5.2 Design Analysis

This section first breaks down the performance and analyzes the contributions of the main designs. It then evaluates the impact of FlacIO on different dimensions of the system (including network, storage space, file access performance, memory, and incremental loading). To simplify, we only report the results of two representative container images: Postgres and Pytorch, which represent small (MB-level) and large (GB-level) images, respectively.

5.2.1 Performance Breakdown

FlacIO consists of two core designs: the runtime image is used to reduce the network overhead of cold startup, and the RTPC is used to provide a lightweight host-side I/O stack. To break down their respective contributions, we evaluate the overhead of the network and local I/O stack by timing the associated functions in FlacIO. Since the network and I/O stack overheads of other systems are difficult to break down (requiring code changes), we merge the two parts together in their results (denoted as image loading).

Figure 7 shows that FlacIO reduces Postgres and Pytorch image data loading overhead by up to 49% and 85% compared to other tested systems. We believe that the main benefit is

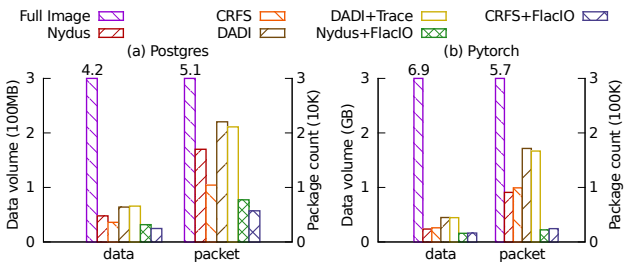

Figure 9: Network Overhead.

that runtime image improves network access efficiency by reducing data and traffic on the network (see §5.2.3). At the same time, the overhead of RTPC (*i.e.*, I/O stack) accounts for only about 1.41% of the overall overhead on average, because the root file system only needs to be built by copying the runtime image to the kernel and associating it with the metadata of the file system. This proves that RTPC achieves the design goal of a lightweight I/O stack.

5.2.2 Factor Analysis

We incrementally enable probe-based tracing and RTPC to show the difference in the benefits of simple exact I/O loading and using runtime image. Three comparison objects include: native Nydus/CRFS (Base), only the probe-based tracing is enabled (Base+PTrace), and both probe-based tracing and RTPC are enabled (Base+PTrace+RTPC). Figure 8 shows the results. In Postgres/Pytorch, the probe-based tracing and the RTPC reduce the latency by more than 8.6%/22.8% and 24.2%/50.0%, respectively. The results prove that only using trace for accurate loading is effective, but not enough. Because this still causes a large number of network I/Os and long I/O paths. The addition of RTPC enables efficient aggregation of network I/O and rapid construction of the root file system.

5.2.3 Network Overhead

To evaluate the network behavior of each system, we use the `tcpdump` to capture network requests during container cold startup and analyze the data volume and the number of packets. Figure 9 shows the results. Take Pytorch as an example, although existing lazy loading solutions reduce the data volume and the number of packets by 29 and 6.3 times compared to the full image loading solution, they are still at least 1.6 and 4.4 times that of FlacIO.

The main reason is that the runtime image ensures that only the required data is loaded (at the page level) when the container is started. However, other lazy loading systems have different I/O amplifications due to the mismatch between the access granularity and lazy loading granularity. In particular, although DADI+Trace prefetches data by replaying the I/O trace collected during the previous startup, it does not significantly reduce the amount of data loaded. For instance, DADI+Trace incurs 1.6 and 1.6 times more data volume than FlacIO in Postgres and Pytorch containers, respectively. This is because the replayed I/Os are not aggregated efficiently and

Table 3: Space Overhead of Runtime Image

	CRFS Img. (Compressed)	Nydus Img. (Compressed)	RT Img.	% (CRFS/Nydus)
Nginx	69.7MB	94.5MB	4.3MB	6.2%/4.6%
Httpd	64.7MB	88.5MB	2.9MB	4.5%/3.3%
Redis	51.9MB	71.8MB	5.1MB	9.8%/7.1%
Memcached	33.6MB	45.3MB	2.8MB	8.3%/6.2%
Tomcat	213.4MB	262.5MB	34.1MB	16.0%/13.0%
Wordpress	237.4MB	321.5MB	19.1MB	8.0%/5.9%
Postgres	152.1MB	210.9MB	16.2MB	10.7%/7.7%
MySQL	178.1MB	247.6MB	31.8MB	17.9%/12.8%
Pytorch	3348.5MB	4229.1MB	146.5MB	4.4%/3.5%
Total	4349.4MB	5571.7MB	262.8MB	6.0%/4.7%

its trace mechanism cannot accurately trace the I/Os of the service in the container. In FlacIO, data is stored continuously in a runtime image, allowing them to be loaded with a single large I/O. In addition, the probe-based I/O tracing mechanism enables FlacIO to accurately track the I/O behavior of services in containers, which also helps reduce network traffic.

5.2.4 Storage Space Overhead

The runtime images are stored in the registry node and bring extra storage space overhead. We evaluate this overhead of 9 popular container services on two mainstream lazy loading systems (CRFS and Nydus). Table 3 shows that the runtime images account for 6% and 4.7% of the total size of the base images (compressed) in CRFS and Nydus, respectively. We consider it cost-effective to pay around 5% of storage overhead for up to 2.4 times of startup latency reduction, as host-side compute resources tend to be more expensive than back-end storage resources on commercial clouds. In addition, the storage overhead of the runtime image can be further reduced by compression, which we put into future work. This experiment also shows the amount of data required to start up the different container services, which represent no more than 18% of the traditional image size.

5.2.5 RTPC Performance

We sequentially perform `read` and `memcpy-after-mmap` on a 1GB file with various I/O sizes to evaluate the impact of stacking the RTPC over the VFS page cache. There are three access scenarios considered in this experiment: direct access to the VFS page cache (*w/o RTPC*), direct access to the RTPC (*RTPC hit*), and access redirected from the RTPC to the VFS page cache (*RTPC miss*). Figure 10 shows that the impact of the RTPC on file access is slight regardless of I/O size and cache status. For example, *RTPC miss* is 5% less than *w/o RTPC* with all I/O size on random read. With `mmap`, *RTPC hit* and *RTPC miss* incurs an averaged 1.7% and 4.6% performance difference with *w/o RTPC*, respectively. This is because the built-in page index of the runtime image is efficient, which ensures high performance of file access on RTPC. For requests that are not hit in the runtime image, RTPC can simply redirect them to VFS. This experiment

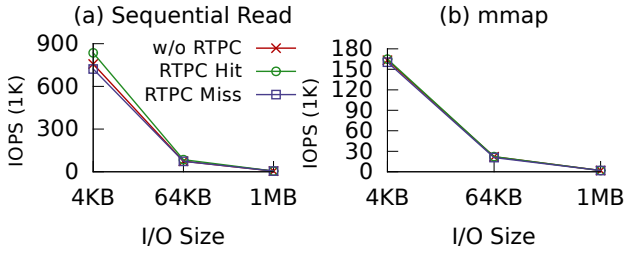

Figure 10: File Access Performance.

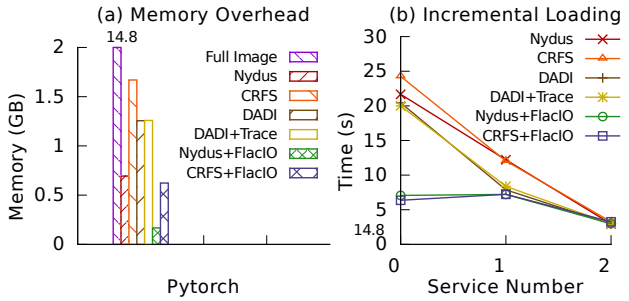

Figure 11: Memory Footprint and Incremental Loading.

shows that RTPC has little impact on file access.

5.2.6 Memory Footprint

We use `vmstat` to profile the memory footprint of all tested systems. Figure 11(a) shows that Nydus+FlacIO has the lowest memory footprint, only 1.1% to 24% of other competitors. There are two main reasons for this result. First, it benefits from the small I/O amplification of FlacIO, *i.e.*, only the necessary pages for startup are loaded. Second, existing lazy loading systems keep data in their internal cache to reduce network access, resulting in the double caching problem (VFS page cache and lazy loading cache). In particular, although the memory footprint of CRFS+FlacIO is also lower than that of other systems, it is much higher (3.7 times) than Nydus+FlacIO, mainly because CRFS itself introduces a high memory footprint. Compared with the original CRFS, CRFS+FlacIO reduces memory usage by 2.6 times.

5.2.7 Incremental Loading

Because runtime images are service-based, there may be a lot of data duplication between runtime images generated from the same base image. FlacIO uses the incremental loading mechanism to solve this problem. We use the native Pytorch as the base image and then run three different services on it. For simplicity, only different Python libraries are imported into the entrypoints of the three services: the first service imports only `torch`, the second service imports `torch` and `triton`, and the third service imports `torch`, `triton`, and `numpy`. We then directly use the entrypoints of these three services as the tracing probes for the runtime image generation. The logical size of the three runtime images is 146MB, 405MB, and 420MB, respectively.

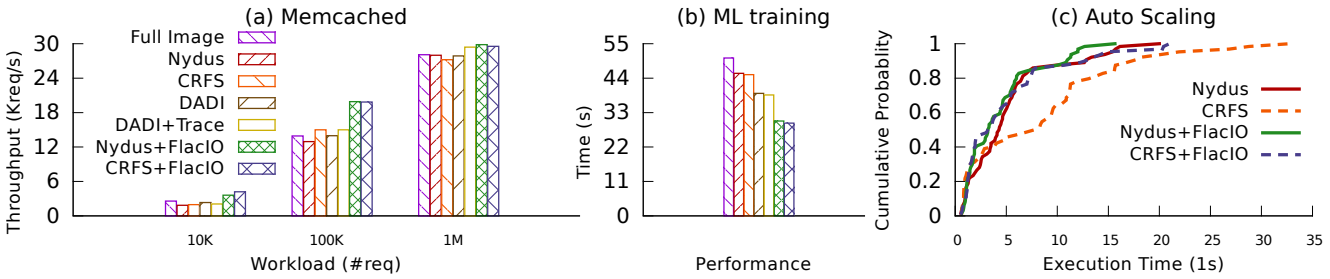

Figure 12: Performance of Real-world Applications.

We start the three services in sequence and evaluate their cold startup latency. Figure 11(b) shows that the startup latency of CRFS/Nydus+FlacIO does not increase as the runtime image increases, because incremental loading only loads nonexistent pages and skips those duplicates. Compared with other lazy loading systems, FlacIO has a significant advantage in the first service, while the advantage of the second and third services is reduced. Unlike the first service, which needs to access a large number of files with small I/Os, the second service only needs to access a large file, and the third service has a small I/O volume.

5.3 Real-World Application

We evaluate FlacIO in three typical application scenarios in the cloud, including object storage, machine learning (ML) training, and dynamic expansion of database. In addition, the runtime images used in this section are the same as in the previous experiments.

5.3.1 Object Storage

We start the Memcached container from the cold state and evaluate the performance of object insertion. The benchmark inserts 10 thousand, 100 thousand, and 1 million 2KB objects into the Memcached. Figure 12(a) shows that the throughput of CRFS+FlacIO and Nydus+FlacIO is up to 2.2 and 1.9 times that of other systems. This is because existing lazy loading systems trigger on-demand image loading when objects are inserted, which affects end-to-end throughput. At the same time, the throughput of other systems can approach FlacIO-based systems only when the size of the dataset increases to 1 million. For example, CRFS+FlacIO(Nydus+FlacIO) achieves 8.6% (6.6%) higher throughput than CRFS (Nydus) at 1 million workloads. This experiment shows that FlacIO can make object storage systems on the cloud warm up faster, which is an important evaluation indicator for cloud products.

5.3.2 ML Training

We evaluate FlacIO in ML training scenario by using the Keras [16] to train the MNIST dataset [24] in the Tensorflow container. The container is started from the cold state and the training dataset is stored in the local file system of the host node and mounted to the container when the service is started. Figure 12(b) shows that training on FlacIO-based systems is 70.1%, 52.1%, 53.5%, and 32% faster than the full image loading, CRFS, Nydus, and DADI, respectively. The reason is that a large number of library files need to be loaded during ML training. As a result, on-demand loading occurs frequently on the lazy loading system. For the FlacIO-based systems, the runtime image already contains most of the image data needed for training, and they are loaded in a single large network I/O when the container starts.

5.3.3 Auto-Scaling

Elastic microservices and serverless computing rely on automatic scaling as a fundamental capability, while container cold startup determines the efficiency of scaling. This section examines the cluster-wide scaling performance of different systems. We utilize a cluster of eight nodes, each scaling up eight Postgres containers. We measure the readiness time of each container, and plot the cumulative distribution function (CDF) results of these container readiness times in Figure 12(c). CRFS has a clear long tail, indicating its very slow cold startup time, while FlacIO-bsed systems have much better tail latency. Moreover, considering the total time to finish the scaling, CRFS+FlacIO(Nydus+FlacIO) is 55.1% (22.5%) faster than CRFS (Nydus). Thus, the overall performance of FlacIO is superior to other systems, confirming our efficiency benefits from the cluster-wide scaling.

6 Conclusion

Container image service is an important component in the cloud computing system. We analyze the shortcomings of existing solutions and conclude that the existing image abstraction is the root cause of the high network overhead and high I/O amplification of container cold startup. Based on this, we propose a new image abstraction called runtime image, which delivers the advantages of efficient network transfer and fast root file system construction. Further, we design FlacIO, a flat and collective I/O accelerator that includes an efficient runtime image structure and a lightweight I/O stack to optimize the container image service. The evaluation shows that the performance of FlacIO is significantly improved compared with the existing systems in many scenarios.

Acknowledgments

We thank our shepherd Yue Cheng and the anonymous reviewers for their constructive comments and feedback. We also thank our colleagues in the Huawei OS Kernel Lab for their support. Yuxin Ren is the corresponding author.

References

[1] Kristen Carlson Accardi. Enabling docking station support for the Linux kernel. In *Proceedings of the Linux Symposium*, 2006.

[2] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2020.

[3] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. Starlight: Fast container provisioning on the edge and over the WAN. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[4] Containerd. Containerd. https://containerd.io/, September 2024.

[5] CRIU. CRIU. https://criu.org, September 2024.

[6] DADI. DADI Trace. https://github.com/containerd/accelerated-container-image/blob/main/docs/trace-prefetch.md, September 2024.

[7] Dragonfly. Dragonfly. https://github.com/dragonflyoss, September 2024.

[8] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[9] eBPF. eBPF. https://ebpf.io/, September 2024.

[10] estargz. estargz. https://github.com/containerd/stargz-snapshotter, September 2024.

[11] Alexander Fuerst and Prateek Sharma. FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[12] Xiang Gao, Mingkai Dong, Xie Miao, Wei Du, Chao Yu, and Haibo Chen. EROFS: A compression-friendly readonly file system for resource-scarce devices. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[13] Gideon Glass, Arjun Gopalan, Dattatraya Koujalagi, Abhinand Palicherla, and Sumedh Sakdeo. Logical synchronous replication in the tintri vmstore file system. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[14] Google. CRFS. https://github.com/google/crfs, September 2024.

[15] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[16] Keras. Keras. https://keras.io/, September 2024.

[17] Kubernetes. Configure liveness, readiness and startup probes. https://kubernetes.io/docs/tasks/configure-pod-container/, September 2024.

[18] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI block-level image service for agile and elastic application deployment. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[19] Huiba Li, Zhihao Zhang, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, Yiming Zhang, and Windsor Hsu. Block-level image service for the cloud. *ACM Transactions on Storage*, 20(1), 2024.

[20] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2022.

[21] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. FlashCube: Fast provisioning of serverless functions with streamlined container runtimes. In *Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS)*, 2021.

[22] Yubo Liu, Yuxin Ren, Mingrui Liu, Hongbo Li, Hanjun Guo, Xie Miao, Xinwei Hu, and Haibo Chen. Optimizing file systems on heterogeneous memory by integrating dram cache with virtual memory management. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2024.

[23] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 115–130, 2003.

[24] MNIST. MNIST dataset. https://yann.lecun.com/exdb/mnist/, September 2024.

[25] Nydus. Nydus. https://nydus.dev/, September 2024.

[26] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[27] openEuler. openEuler 22.03 LTS. https://www.openeuler.org/en/download/archive/detail/?version=openEuler%2022.03%20LTS, September 2024.

[28] OverlayFS. OverlayFS. https://docs.kernel.org/filesystems/overlayfs.html, September 2024.

[29] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-Grained isolation for scalable, dynamic, multi-tenant edge clouds. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[30] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 341–354, 2021.

[31] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[32] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[33] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast in-memory CRIU for docker containers. In *Proceedings of the International Symposium on Memory Systems (MemSys)*, 2019.

[34] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNET: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2021.

[35] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[36] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2023.

[37] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2006.

[38] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[39] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt1. DupHunter: Flexible high-performance deduplication for docker registries. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[40] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018.